Buletinul Stiintific al Universitatii "Politehnica" din Timisoara, ROMANIA
Seria AUTOMATICA si CALCULATOARE
PERIODICA POLITECHNICA, Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE
Vol.49 (63), 2004, ISSN 1224-600X

An attempt to set the framework of Model-Oriented Programming

Philippe Lahire* Didier Parigot** Carine Courbis*** Pierre Crescenzo* Emanuel Ţundrea****

* Laboratoire I3S (UNSA/CNRS) - Projet OCL - 2000 route des Lucioles - Les Algorithmes, Bâtiment Euclide - BP 121 - F-06903 Sophia-Antipolis cedex - France
{Philippe.Lahire, Pierre.Crescenzo}@unice.fr
http://www.i3s.unice.fr/

** INRIA Sophia-Antipolis - 2004, route des Lucioles - BP 93 - F-06902 Sophia-Antipolis cedex - France
Didier.Parigot@inria.fr
http://www.inria.fr/

*** University College London - Computer Science Department - Adastral Park - Martlesham IP5 3RE - United Kingdom
Carine.Courbis@bt.com
http://www.cs.ucl.ac.uk/

*** "Politehnica" University of Timisoara - Faculty of Automatics and Computer Science - Bd. V. Parvan no 2 - 1900 Timisoara- Romania
Emanuel@emanuel.ro
http://www.utt.ro/

***Abstract*— Nowadays, companies involved in the development of modern software face several difficulties. One of the most important ones is the continuous evolution of software platforms (C++, Java, DotNet, CORBA, EJB, Web services, XML, etc.). One interesting solution to this problem is the Model-Driven Architecture (MDA) approach from the OMG. It suggests that domain-specific knowledges should be encapsulated in platform-independent business models, apart from the applications. Beyond this answer is the failure of classical development techniques that rely on object-oriented design and programming. According to these remarks, we address another way to develop software: *Model-Oriented Programming*. It is based on the Domain-Driven Development track and introduces a macro-level on top of the classical programming entities. It intends to be used for the handling, reuse and evolution of the business know-how and its associated applications. This paper attempts to define a set of golden rules for setting up the framework of model-oriented programming and ensuring the success of its use. It gives also an overview of the implementation of those rules that we propose in our approach called SmartModels.**

***Index Terms*— Meta-modelling, Business models, Model-Driven Architecture (MDA), Domain-Driven Development (DDD), aspect-oriented programming (AOP), Generative programming**

I. Introduction

The way an application must be developed must evolve in order to take into account new trends in order to provide softwares which are more open, adaptable and evolutive. Main reason is that with the continuous evolution and the proliferation of new technologies it is difficult to choose the right and more capable of evolving one and most of all classical approaches do not provide the right answer to allow a company to quickly and cheaply adapt its software to new user needs and technologies. The lacks of object-oriented programming explain the emergence of new programming paradigms such as AOP (*Aspect-Oriented Programming*) [16], SOP (*Subject-Oriented Programming*) [15], IP (*Intentional Programming*) [24], or component programming [27]. At the specification level, a strong and continuous evolution is undergoing toward standards of the W3C (*World Wide Web Consortium*) for documents or of the OMG (*Object Management Group*) for design methodologies such as UML (*Unified Modeling Language*) or MDA (*Model-Driven Architecture*) approach [28], [4], [14].

According to these observations, we propose another way of developing software, named *Model-Oriented Programming*. It is based on the Domain-Driven Development track (DDD) [9], which relies on several paradigms such as object-oriented technology, languages for components, MDA, approaches for the separation of concerns, and generative programming [8].

Model-oriented programming is a new approach for the development of software which takes advantage not only from object-oriented and aspect-oriented paradigms but also from information systems. As it has been mentioned above, model-oriented programming introduces a new level of abstraction (the model) which acts as an autonomous entity that may receive queries from satellite applications. The specification of both the model and the applications may use for example object-oriented and/or aspect-oriented approaches.

Each application is built around at least one business model. We address the most frequent case where one business model is predominant. Having a predominant model on which are plugged in different concerns of an application is very similar to approaches by separation of concerns (ASoC). However, in model-oriented programming, concerns are attached to a business model instead of being weaved into object-oriented applications which may be executed with or without these concerns. The model has its behaviour (its semantics), but it does not invoke itself any treatment. On the contrary, the semantics of the model is addressed only when applications query the model entities in order to match their requirements. In the context of DDD, a business model may support two main categories of applications: i) those dedicated to the computation and/or the update of information recorded by the instances of models; their methodology is close from information systems, and ii) those which deals with the transformation of the model and which are particularly relevant in the context of MDA.

Model-oriented programming is definitively very different from other paradigms such as object-oriented programming (OOP). It breaks the supremacy of programming languages: the model is now the key-point whereas the formalism used to describe its instances play minor roles. This is the consequence of the collaboration between MDA and generative programming. Altogether these two paradigms contribute to link the model and its formalism(s), and this favours the coming out of Domain-Specific Languages (DSL) as business-models.

A new approach for the development of software must ensure that software engineering skills are covered and improved in comparison with object-oriented and aspect-oriented approaches. Reusability, evolution capabilities and robustness of both business models and applications must be addressed very carefully by model-oriented programming. We propose nine rules which characterize, from our point of view, model-oriented programming; they are classified in two categories: conceptual and implementation purposes. These rules relies on the experience gained in previous works which deal on the one hand with meta-modeling [7], and on the other hand with the design of a software factory called SMARTTools [2]. They intends to take advantage of both approaches in order to propose a framework for the development of domain-specific applications. In sections II and III we address rules related respectively to the design of the meta-model and its implementation. Section IV deals with the description of a possible implementation of those rules in an approach called SMARTMODELS and section V mention some of the related works. finally we conclude.

II. RULES FOR THE APPROACH DESIGN

Rule N°1: Business Model as a first-class entity of the development process. : A business model relies on a data model and on a semantic model. The data model contains the description of the entities involved in the business model whereas the semantic model describes the interactions and the constraints between those entities, but also their behaviour with respect to the business-model know-how. A business model is considered by applications as a whole or for its contents; it constitutes a new level of abstraction which favours global operations such as transformation or introspection. Both of them query the model entities in order to reuse its business know-how or to involve both model and programs.

Rule N°2: A triple independence between the model, the application and the technology.: A business model is not an application. It encapsulates the description of its behaviour (its semantics), which must be independent from any further use. This property will ensure that a business model is reusable independently from the applications that may address it[1]. Moreover, an application or a business model must be designed independently from the software platform on which the application will be executed. This is only at the very last moment that the binding with the platform technology[2] must be made. This second property allows the business logic to be used whatever technology will appear in the future.

Rule N°3: Support of generic entities. : Typically business models may address lines of products and more generally a set of entities that may have commonalities and differences but which have a close semantics[3]; they must be designed as generic entities which may be easily derived. A quite common situation is that business models address a few key-entities which are defined according to a large number of basic entities; very often, the key-entities correspond to generic entities. Then it is particularly important that generic entities provide a clear vision of their semantics because they deal with a significant part of the model semantics. Object-Oriented languages like Eiffel proved that the support of generic entities (we should say generic business model) is an interesting approach to ensure reusability and evolution.

Rule N°4: Clear separation between semantic and data models.: The domain-specific know-how is encapsulated in business models through the data model (reification and structuring by the entities) and the semantic model (behaviour of those entities). To be able to reuse the semantics when the data model evolves is an important issue. This is particularly important in the context of model transformations where semantics must evolve accordingly to the data-model (in the most automatic way). Model-oriented programming must provide a clear separation between the description of the data-model and the description of its semantics.

[1] Of course it is still an important issue to ensure also the reusability of the application behaviour.

[2] Some people call this the Platform Dependent Model (PDM).

[3] It is important to note that commonalities and differences may represent a major part of the semantics of these entities.

Rule N°5: Orthogonal handling of concerns.: Rule N°2 infers a separation of concerns between the business model and the applications. The first one is under the responsibility of an expert which captures the domain-specific know-how, whereas the second one is handled by programmers. But separation of concerns must exist also within the business model and within the applications themselves.

According to the business model, the needs are twofolds: *i)* the semantics may be complex enough and require some modularization, and *ii)* pieces of semantics which are orthogonal to the original semantics must be straightforwardly carried out.

According to applications, the requirements are even more important. An application may contain different subjects which have to be smoothly composed for building it up. Moreover, an application should be able to take care about the evolution of the environment (which can not be foreseen in advance), without changing the application core.

III. Rules for the Approach Implementation

Rule N°6: An adequate balance between declarative and imperative programming.: Semantics of business models should be described as much as possible in a declarative way in order to specify what is expected (the "*what*") but not how it is made (the "*how*"). This is one of the most important issues addressed by the MDA approach. But, it is not acceptable to carry this approach to the breaking point where the description relies on very complex formalisms, difficult to read and to understand. A compromise is necessary between the "all declarative" and the "all programming".

Rule N°7: Support of domain-specific languages.: A clear distinction has to be made between the expressiveness of a business model and the language (textual, graphical, etc.) used by the designer for the specification of the different pieces of this business model. Moreover, model-oriented programming tends to come closer and closer to the general public (ubiquitous programming), so that the need to provide "languages" dedicated to one business model and even to one application becomes more and more important. Generative programming and MDA provide a good support to achieve this issue.

Rule N°8: Openness of the development process.: To provide a meta-model and a set of related mechanisms that answer to any need of any kind of business model is Utopian from our point of view. We promote the idea of an unified approach with very few built-in mechanisms, but that can be easily adapted to further needs of modern applications. In particular, it is important to be able to customize the way to query information according to the context of use. In other words, the generation and handling of an executable business model must be customizable. In our approach, all the key-concepts which participate to the description of both the application and the business model in order to make it executable are first-class entities.

Rule N°9: Self-extensible capability of the approach.: Model-oriented programming requires a meta-model which captures the description of both business models and applications, as it is mentioned in previous rules. This meta-model may be considered as a particular business model. As it is explained in Rule N°7, the specification of the different parts of this meta-model may rely, for example on a dedicated language[4]. But many other needs required for the development of applications may appear. In particular, modern applications should be available as components that may interact one with the others. It is important to make the approach self-extensible, that is to say able to include other applications and business models built thanks to model-oriented programming (that means built with the approach itself). For example, to handle components, a correct approach would be to design a business model.

With those nine rules, we attempted to set a framework for model-oriented programming. We promote the idea that an approach which intends to implement model-oriented programming should try as much as possible to match the requirements proposed by those rules. In the next section we propose some elements of response addresses the rules.

IV. Key-aspects of business models with SmartModels

This section deals with a subset of the rules described in previous sections and tries to explain how to map those rules in the meta-model associated to our approach which is called SmartModels and which allows to describe business models (reification and semantics). But we do not address here the aspects related to the modelisation of applications dedicated to these business models.

As a preamble, we can say that the meta-model which allows the description of business models addressed *i)* the reification of basic entities (Section IV-B), *ii)* the reification of generic entities and their generic parameters (Section IV-C), *iii)* the semantics of the business model which corresponds mainly to the possible values that may be assigned to the generic parameters (Section IV-C) and to the actions (Section IV-D). One of the key-aspects of SmartModels is that the semantic model is encapsulated in a meta-level, so that it may be distinguished from the data-model. Section IV-A explains the main benefits that are expected from this. Figure 1 illustrates these previous lines.

A. A meta-level to separate semantic and data-models

This section addresses principally the fourth rule. In Figure 1 we propose an overview of the architecture of the meta-model. The semantics of the business model is addressed through the specification of *hypergeneric parameters*[10], *characteristics* and *actions*. All of them participate to the definition of the semantics of business-model entities[5] (whether they are generic or not); but they do not address the description of their instances. Because applications are outside the business model, the methods that handle instances of atoms are accessors only[6]; they are automatically generated

[4]It can be built as a pseudo-language or it can use the UML graphical approach with activity or class diagrams, Action Semantics, etc.

[5]We call them *atom* - see section IV-B.

[6]This is the main difference with actions which address entities but not instance of entities.

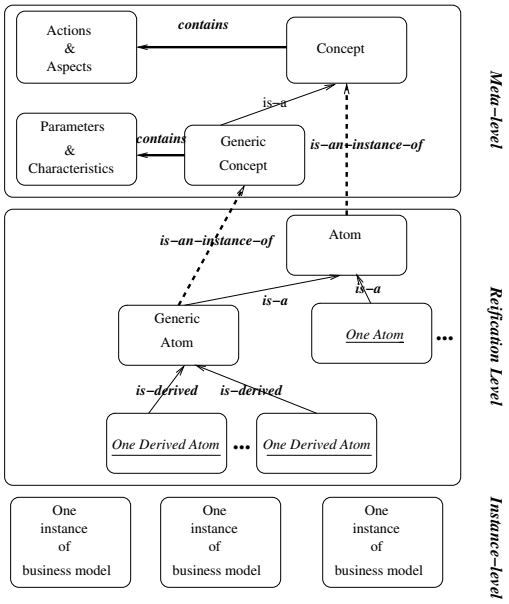

Fig. 1. Key-aspects of a business model

taking into account the type (for example if it is a collection or not). *Actions* are methods with special properties; for example, they can handle assertions (see section IV-D) and aspects (not presented in this paper). Moreover, actions are first-class entities and are integrated in a meta-object protocol.

We propose to create a meta-level in order to encapsulate the semantics of an entity into a meta-level which is named the *concept*. A concept is associated with one or several atoms[7]. This clear separation between the semantics of the business model and the reification of its entities is very important because it favours *i)* the maintenance of the semantics (redefining the semantics should only deal with concepts), *ii)* the reuse of the semantics in other (closely-related) business models, and *iii)* the transformation of model which is one of the key-points of model-oriented programming. MOF does not integrate any meta-level. The main consequence is that it is not possible[8] to indicate that a MOF class is an instance of another one. The main facilities provided by MOF to describe meta-information are class variables and class methods; from our point of view it is not sufficient.

B. Expressiveness of the data-model

This section addresses mainly the first and the second rules; it participates to the description of the data-model which is part of a business model.

The description of the business-model entities relies on well-known concepts that may be found in most programming languages or meta-models. We present them briefly in the context of SMARTMODELS and with regard to MOF [13]. At a first glance, we could define business models directly with MOF, but Sections IV-A and IV-C demonstrate that additional information must be inserted.

In our meta-model, an *atom* is the structure which supports the description of an entity; it is very close to the MOF "*class*" notion[9]. Then the features provided by MOF to describe the contents of a class (such as attributes, operations, generalisation relationships) are sufficient to define most of the reification of an entity. MOF provides also the possibility to describe associations. To describe associations, we have introduced a generic type which implements different kinds of collection such as *bag*, *set*, or *list* (with or without bounds)[10].

The designer of a business model may create atoms either for improving the structuring and factorization of information within the model hierarchy, or for describing atoms which have instances within applications. Our meta-model provides a way to address those two issues; MOF does it through the notion of *abstract class*. If it means that the class must have at least an abstract method or that all the methods must be abstract, then we believe that this mechanism is not sufficient. In particular, some applications may be interested by some atoms whereas others are not; it is not the same thing to say that whatever is the context of use, one atom may not have instances because it is only partially defined. We believe that a more accurate information according to the atom status will improve the readability of the code produced by generators, and the facilities that may be provided or not to the programmer of application according to it. The interest to be able to associate different status with an atom is even greater if the business model may import atoms from another business model.

C. Support of generic entities

This section addresses mainly the first, third and fourth rules. It participates to the description of the data and semantics models; it addresses especially the handling of the key-entities of a model.

The support of generic entities (*generic atoms*) is an important issue for business models. Let us take an example of one business model which is dedicated to record both the structures and semantics of Java programs. Possible applications with respect to this model may implement functionalities of programming environments (metrics, various wizards or editors, etc.). Possible atoms of this model represent, for example, *attribute*, *method*, *method parameters*, *modifiers*, etc. But the most interesting ones deals with the different kinds of *classifiers* and *relationships* (aggregation-like or inheritance-like). Most semantics may be encapsulated within classifiers and relationships and other atoms mentioned above may have a very minimal semantics mostly represented by their reification. This is possible because they are driven by the semantics associated with classifiers and relationships. In fact, there are

[7]It is an approach which is quite similar to the classes and meta-classes of the Smalltalk language.

[8]It is of course possible to write two models with MOF, one being the meta-model of the other. But according to our knowledge is not possible to express it with MOF.

[9]The concept of *class* is, from our point of view, too much related to programming languages whereas business models require a more abstract concept.

[10]The MOF associations provide more capabilities but we are not sure at the moment that business model description requires it.

several kinds of classifiers (e.g. *class*, *inner class*, *interface*, etc.) and relationships (e.g. *extends* between interface, *extends* between classes, *implements* between one interface and one class) in this business model [7]. Then it is meaningful to be able to record their definitions as generic atoms[11].
Let us define the term "*generic atoms*". The genericity comes from a set of *hypergeneric parameters* and a set of *characteristics* which records the differences and the commonalities between all the foreseen derived entities[12] (*e.g.* all the Java classifiers). The definition of an hypergeneric parameter is mainly based on a basic type (it may be an integer, a boolean, an enumeration, a tuple and a collection) and on some additional pieces of information. The definition of a characteristic relies on an atom or a collection of atoms (*e.g.* one kind of classifier records the possible kinds of inheritance-like relationships that it may declare). Intuitively, generic atoms are quite similar to the concept of generic class in the Eiffel language. But derived atoms are obtained through the relevant combination of values associated with the sets of characteristics and parameters which participate to the definition of the *generic atom*.

We choose to use generic atoms instead of inheritance relationships for modeling these atoms for several reasons: *i)* the definition of the data model is not mixed with the definition of the semantics; this increases the ability of the semantics of the business model to be transformed and reused in another model; *ii)* a significant part of the semantics of all the derived entities of one generic entity is recorded in one location, by the definition of its parameters and characteristics; this favours both (re)use and maintenance of these entities; iii) the reuse of a business model is improved, especially when the new model is an extension of the first one; according to the business model related to the Java language, to extend it with a new relationship like for instance, the reverse-inheritance requires only that the new business model describes a new instance of the generic entity which deals with inheritance-like relationships.

D. Description of the semantic model

This section addresses the first rule and more especially the description of the semantics model.

We explained in Section IV-C that a significant part of the semantics of a business model is encapsulated in a few generic atoms. A part of the semantics is captured by parameters, characteristics and invariants[13]. It is a first step but it is still not sufficient to handle the full semantics of atoms. For example, the value of parameters used for the instantiation of generic atoms will affect the behaviour of its derived atoms. It is necessary to be able to specify this behaviour.

Each atom, whatever it is generic or not, has a meta-level (its concept) where it is possible to define actions; when the atom is derived from a generic one its execution is driven by the value of the parameters. For other atoms, the ability is provided but we believe that it is not relevant in most cases.

An *action* has a signature, preconditions and postconditions defined with respect to the reification of entities and hypergeneric parameters when it is a generic atom[14]. It may also accept the execution of orthogonal concerns (aspects). An action must be completely independent from the application related to the business model. A typical scenario is that the behaviour of a given application relies on the semantic model, that is to say call those actions or query hypergeneric parameters.
It is straightforward that an action has a "body" that has to be specified; there are three main approaches to take its contents into account in the meta-model: *i)* to propose a full representation of the body which may correspond to the reification of some pseudo-languages, *ii)* to delegate to the description of the body to the underlying implementation language; the meta-model records only the fact that an action has a body, and *iii)* to propose a partial description of the body[15]. Typically for the first and third solution the body of the action will be partially generated, whereas in the second solution, the whole description of the body will be completely under the responsibility of the developer. At the moment our first prototype implements the third solution but the expressiveness of action bodies is going to be improved as far as interesting capabilities are found.
It is important to distinguish the capability of the model to record more or less the representation of the action bodies, from the description language which is provided to the user in order to describe it. About this aspect two solutions seem to be relevant: *i)* to take UML from OMG and to use diagram of activities and/or Action Semantics, *ii)* to design a domain-specific language for the semantics description. We have not evaluated seriously these two solutions yet. It is said that UML is the meta-model for the specification of business models (it is important to reuse existing standards), but we have also to remember that at the beginning UML was not designed for the definition of business models but for applications. An alternative to those approaches may be to increase the expressiveness of MOF with respect to the description of the semantics.

V. RELATED WORK

Several approaches address the problematic suggested in the introduction, that is to say : to capitalize the business know-how and its associated applications in order to handle the continuous evolution of software platforms.

The AOP-related works try to propose powerful mechanisms to describe the semantics of domain specific languages (DSLs) [3]. All of these works [18], [22], [23], [17] stem from the basic issue of a better separation between the data structure and the semantics treatments.

It is well-known that the handling of an AOP can be rather complex and can introduce scarcely controllable situations [5].

[11]One generic entity for modifiers, one for inheritance-like relationships and one for aggregation-like relationships.

[12]This is the term which is quite often used in the state of the art, to refer instances of generic entities.

[13]Like in MOF or UML, it is possible also to define atom invariants. This contributes to the description of the semantics of entities.

[14]In our meta-model, assertions are described with OCL from the OMG.

[15]For example, to record the list of hypergeneric parameters that are involved in the semantics of the action

To solve this problem, aspect-oriented languages dedicated to the context are proposed [25]. However, nearly in all the cases, the reflexivity mechanism plays major role [20], [19]. Because of this, from our point of view, there is a strong dependence between the approach and the implementation techniques (that should be as less visible as possible at the model level).

Some approaches focused on the issues of modularity or reuse of semantics components [3], [26]; Other investigate how to introduce powerful mechanisms to reuse language components; their objective is to be able to design a DSL by composition of existing components [3]. Model-oriented programming is more oriented toward the definition of a family of DSLs.

About Modelling approaches, Model-oriented programming is closer to those that advocate a domain model approach [1] than those that propose extensions (profiles) of a standard model. Indeed, having an universal model seems no longer be the solution advocating by the MDA but rather an approach à la MDA.

With respect to model transformation approaches (MDA) [1], the coupling of model-oriented programming with generation mechanisms allows much more complex transformation than those offered by simple transformations between models[4], [1]. But foreseeing a translation of the treatments from the original model to the target model [21] is important.

Finally, Model-oriented programming, is integrated in the much more global context of software factories [12], [6], [11].

VI. Conclusion and perspectives

In this paper, we propose to structure the framework of Model-Oriented Programming with a set of essential rules. We consider them as a first attempt for the definition of the main principles of this approach. We are working on an approach called SmartModels, for which some of the key-aspects had been presented. It is one interpretation of those rules and an implementation of this approach on top of SmartTools [2] is undergoing implementation.

In the short term we want to experiment our approach for the description of various business models and their applications; currently we start to investigate business models of a the French electricity company, EDF. The objective is to get feedbacks in order to improve the expressiveness of SmartModels but also the automation of *i)* the generation of the behaviour, and *ii)* the semantics transformation of both business models and applications when they evolve toward another model or application.

References

[1] Colin Atkinson and Thomas Kühne. The role of meta-modeling in MDA. In Jean Bezivin and Robert France, editors, *Workshop in Software Model Engineering*, 2002.

[2] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, JoÃ«l Fillon, Didier Parigot, Claude Pasquier, and Claudio Sacerdoti Coen. SmartTools: a development environment generator based on XML technologies. In *XML Technologies and Software Engineering*, Toronto, Canada, May 2001. ICSE'2001, ICSE workshop proceedings. ftp://ftp-sop.inria.fr/oasis/personnel/Carine.Courbis/smarticse02.pdf.

[3] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing Domain-Specific Languages. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 143–153. IEEE Computer Society Press, 1998.

[4] Jean Bezivin. From Object Composition to Model Transformation with MDA. In *TOOLS USA*, Santa-Barbara, August 2001. IEEE TOOLS-39.

[5] Noury M. N. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. In *Technique et Sciences Informatiques*, volume 20, page 505 à 528. HermÃ¨s, 2001.

[6] Steve Cook and Stuart Kent. The Tool Factory. In *OOPSLA'2003, workshop on Generative Techniques in the context of MDA*, Anaheim - USA, October 2003.

[7] Pierre Crescenzo and Philippe Lahire. Using both specialisation and generalisation in a programming language: Why and how? *Lecture Notes in Computer Science*, 2426:64–73, 2002.

[8] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, June 2000.

[9] Krzysztof Czarnecki and John Vlissides. Domain-Driven Development. Special Track at OOPSLA'03 URL: http://oopsla.acm.org/oopsla2003/files/ddd.html.

[10] P. Desfray. *Object Engineering, the Fourth Dimension*. Addison-Wesley Publishing Co., 1994.

[11] Christer Fernström, Kjell-Håkan Närfelt, and Lennart Ohlsson. Software factory principles, architecture, and experiments. *IEEE Software*, 9:36–44, March 1992.

[12] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27. ACM Press, 2003.

[13] Object Management Group. Meta Object Facility (MOF) specification (version 1.3). Technical report, Object Management Group, March 2000.

[14] OMG Staff Strategy Group and Richard Soley. Model-Driven Architecture. Technical report, OMG, November 2000.

[15] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In Andreas Paepcke, editor, *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428. ACM Press, October 1993.

[16] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[17] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[18] Karl J. Lieberherr and Doug Orleans. Preventive program maintenance in Demeter/Java. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 604–605. ACM Press, 1997.

[19] Cristina Videira Lopes and Karl J. Lieberherr. AP/S++: A CASE-study of a MOP for purposes of software evolution. Technical Report NU-CCS-95-?, Xerox PARC and Northeastern University, November 1995.

[20] Jacques Malenfant and Pierre Cointe. Aspect-Oriented Programming versus Reflection: a first draft. In *Position Statement for the OOPLSA '96 AOP meeting*, 1996.

[21] OMG. MDA - Model-Driven Architecture. http://www.omg.org/mda.

[22] Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. *Lecture Notes in Computer Science*, 2192:73–??, 2001.

[23] Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. *Science of Computer Programming*, 29(3):303–326, 1997.

[24] Charles Simonyi. The death of programming languages, the birth of intentional programming. Technical report, Microsoft, Inc., September 1995.

[25] Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *USENIX Conference on Domain-Specific Languages*, 1997.

[26] Y. V. Srinivas and Richard Jullig. Specware(TM): Formal support for composing software. Technical Report KES.U.94.5, Kestrel Institute, 1994. see also Proceedings of the Conference on Mathematics of Program Construction, Kloster Irsee, Germany.

[27] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[28] T Ziadi, B Traverson, and Jean-Marc JÃ©zÃ©quel. From a UML Platform Independent Component Model to Platform Specific Component Models. In *International workshop in Software Model Engineering (WiSME02) at UML2002*, Dresden (Germany), September 2002.